Multithreaded Input-Sensitive Profiling

Emilio Coppa
Dept. of Computer Science
Sapienza University of Rome
coppa@di.uniroma1.it

Camil Demetrescu
Dept. of Computer and System Sciences
Sapienza University of Rome
demetres@dis.uniroma1.it

Irene Finocchi
Dept. of Computer Science
Sapienza University of Rome
finocchi@di.uniroma1.it

Romolo Marotta
Dept. of Computer and System Sciences
Sapienza University of Rome
romolo.marotta@gmail.com

Abstract

Input-sensitive profiling is a recent performance analysis technique that makes it possible to estimate the empirical cost function of individual routines of a program, helping developers understand how performance scales to larger inputs and pinpoint asymptotic bottlenecks in the code. A current limitation of input-sensitive profilers is that they specifically target sequential computations, ignoring any communication between threads. In this paper we show how to overcome this limitation, extending the range of applicability of the original approach to multithreaded applications and to applications that operate on I/O streams. We develop new metrics for automatically estimating the size of the input given to each routine activation, addressing input produced by nondeterministic memory stores performed by other threads as well as by the OS kernel (e.g., in response to I/O or network operations). We provide real case studies, showing that our extension allows it to characterize the behavior of complex applications more precisely than previous approaches. An extensive experimental investigation on a variety of benchmark suites (including the SPEC OMP2012 and the PARSEC benchmarks) shows that our Valgrind-based input-sensitive profiler incurs an overhead comparable to other prominent heavyweight analysis tools, while collecting significantly more performance points from each profiling session and correctly characterizing both thread-induced and external input.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement Techniques; D.2.8 [*Software Engineering*]: Metrics—performance measures

General Terms Algorithms, Measurement, Performance.

Keywords Asymptotic analysis, dynamic program analysis, instrumentation, I/O streams, multithreading, performance profiling, Valgrind, workload characterization.

1. Introduction

Performance profilers collect information on running applications and associate performance metrics to software locations such as routines, basic blocks, or calling contexts [1, 9, 26]. They play a crucial role towards software comprehension and tuning, letting developers identify hot spots and guide optimizations to portions of code that are responsible of excessive resource consumption.

Unfortunately, by reporting only the overall cost of portions of code, traditional profilers do not help programmers to predict how the performance of a program scales to larger inputs. To overcome this limitation, some recent works have addressed the problem of designing and implementing performance profilers that return, instead of a single number representing the cost of a portion of code, a cost function that relates the cost to the input size (see, e.g., [5, 8, 31]). This approach is inspired by traditional asymptotic analysis of algorithms, and makes it possible to analyze – and sometimes predict – the behavior of actual software implementations run on deployed systems and realistic workloads. Some of the proposed methods, such as [8], perform multiple runs with different and determinable input parameters, measure their cost, and fit the empirical observations to a model that predicts performance as a function of workload size. More recent approaches make a step further, tackling the problem of automatically measuring the size of the input

given to generic routines [5, 31], collecting data from multiple or even single program runs.

As observed in [5] and [31], a current limitation of input-sensitive profilers is that they specifically target sequential computations, ignoring any communication between threads. Multithreaded applications based on concurrent programming are traditionally difficult to analyze, since threads can interleave in a nondeterministic fashion and affect the behavior of other threads. Nevertheless, they are widespread in modern multicore architectures, making the quest for dynamic analysis tools for concurrent computations extremely critical.

Our contribution. In this paper we show how to extend the input-sensitive profiling methodology to the full range of concurrent applications, hinging upon the approach described in [5]. The ability to automatically infer the size of the input data on which each routine activation operates is a crucial issue in input-sensitive profiling, but current techniques may fail to properly characterize the input size in a multi-threaded environment. As shown in this paper, if the input size is not estimated correctly, the analysis of profiling data can lead to uninformative cost plots or even to misleading results. As a first contribution we therefore propose a novel metric, called *threaded read memory size*, that overcomes this limitation, addressing input produced by non-deterministic memory stores performed by other threads and by the OS kernel (e.g., in response to I/O or network operations). We provide real case studies, based on the `MySQL` database management system and on the `vips` image processing tool, showing that our extension allows it to characterize the behavior of complex applications more precisely than previous approaches. We then demonstrate that the input size of a routine can be automatically and efficiently computed in a multithreaded setting, and discuss the implementation of a Valgrind-based input-sensitive profiler for concurrent applications (the tool is available at `http://code.google.com/p/aprof/`). An extensive experimental investigation on a variety of benchmark suites (including the SPEC OMP2012 and the PARSEC benchmarks) shows that our profiler incurs an overhead comparable to other prominent Valgrind tools, while collecting significantly more performance points from each profiling session and correctly characterizing both thread-induced and external input.

Paper organization. The remainder of this paper is organized as follows. In Section 2 we introduce the threaded read memory size metric, showing its usefulness through synthetic examples. Case studies drawn from real applications are discussed in Section 3. Section 4 proposes an efficient algorithm for computing the threaded read memory size of a routine activation. Section 5 describes the most relevant implementation aspects and Section 6 presents the results of our experimental evaluation. Related work is discussed in Section 7 and concluding remarks are given in Section 8.

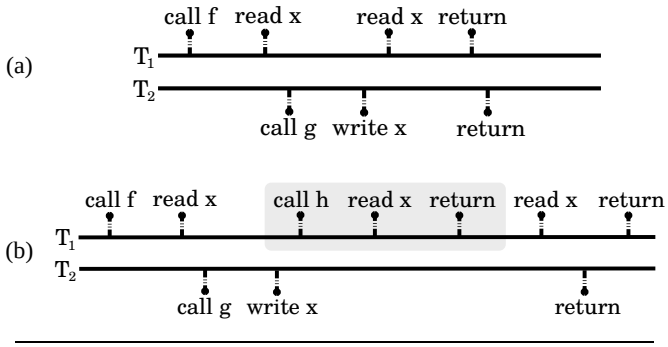


Figure 1. Threaded read memory size examples.

2. Multithreaded Input Size Estimation

A crucial issue in input-sensitive profiling is the ability to automatically infer the size of the input data on which each routine activation operates. This can be done in a single-threaded scenario using the *read memory size* metric introduced in [5]:

Definition 1 ([5])**.** *The* read memory size *(*RMS*) of the execution of a routine r is the number of distinct memory cells first accessed by r, or by a descendant of r in the call tree, with a read operation.*

The intuition behind this metric is the following. Consider the first time a memory location ℓ is accessed by a routine activation r: if this first access is a read operation, then ℓ contains an input value for r. Conversely, if ℓ is first written by r, then later read operations will not contribute to increase the RMS since the value stored in ℓ was produced by r itself.

The RMS, although very effective in single-threaded executions, may fail to properly characterize the input size of routine activations in a multi-threaded environment. Consider, as an example, the execution described in Figure 1a: routine f in thread T_1 reads location x twice, but only the first read operation is a first access. Hence, $\text{RMS}_f = 1$. Notice however that routine g in thread T_2 overwrites the value stored in x before the second read by f: this read operation gets a value that is not produced by routine f itself and that should be therefore regarded as new input to f. The same drawbacks discussed in the above example arise when one or more memory locations are repeatedly loaded by a routine with values read from an external source, e.g., network or secondary storage. To overcome these issues, we propose a novel metric for estimating the input size, which we call *threaded read memory size*.

Definition 2. *Let r be a routine activation by thread t and let ℓ be a memory location. An operation on ℓ is:*

- *a* first-access, *if ℓ has never been accessed before by r or by any of its descendants in the call tree;*
- *an* induced first-access, *if the latest* `write`(ℓ) *performed by any thread $t' \neq t$, if any, has not been followed by an*

procedure `producer()`

```
1: while (1) do
2:     wait(empty)
3:     wait(mutex)
4:     produceData(x)
5:     signal(mutex)
6:     signal(full)
```

procedure `consumer()`

```
1: while (1) do
2:     wait(full)
3:     wait(mutex)
4:     consumeData(x)
5:     signal(mutex)
6:     signal(empty)
```

Figure 2. Producer-consumer pattern: when n values have been produced, $\text{RMS}_{\texttt{consumer}} = 1$ while $\text{TRMS}_{\texttt{consumer}} = n$.

access to ℓ by routine r or by any of its descendants in the call tree.

Definition 3. *Let r be a routine activation by thread t. The* threaded read memory size $\text{TRMS}_{r,t}$ *of r with respect to t is the number of read operations performed by r that are first-accesses or induced first-accesses.*

We notice that the RMS coincides with the number of read operations that are first-accesses and therefore

$$\text{TRMS}_{r,t} \geq \text{RMS}_r \tag{1}$$

for each routine activation r and thread t.

Example 1. Consider again the example in Figure 1a: we have $\text{TRMS}_{f,T_1} = 2$. The first read operation on x is indeed a first-access, while the second one is an induced first-access: between the latest write operation on x performed by thread $T_2 \neq T_1$ and the second `read`(x) by routine f there are no other accesses to x by f.

Example 2. Consider Figure 1b. In this case $\text{RMS}_h = 1$ and $\text{RMS}_f = 1$: function f performs three read operations on x (one of which through its subroutine h), but only the first one is a first-access and contributes to its RMS. With respect to the TRMS_{f,T_1}, the read operation by h is an induced first-access for f (similarly to the previous example), while the third read is not: between the latest write operation on x performed by thread $T_2 \neq T_1$ and the third `read`(x), f has already accessed x through its descendant h.

We also have $\text{TRMS}_{h,T_1} = 1$. Notice that the read operation in h could be regarded both as a first-access and as an induced first-access with respect to h: since we are interested in characterizing communication between threads via shared memory, we will classify accesses of this kind as induced first-accesses.

Example 3. Producer-consumer is a classical pattern in concurrent applications. The standard implementation based on semaphores (see, e.g. [27]) is shown in Figure 2, where producer and consumer run as different threads and routines `produceData` and `consumeData` write to and read from memory location x, respectively (the implementation can be easily extended to buffered read and write operations). For simplicity of exposition, we will not consider memory accesses due to semaphore operations. With this assumption, $\text{RMS}_{\texttt{consumer}} = 1$, since the consumer repeatedly reads the same memory location x. Conversely, the threaded read memory size gives a correct estimate of the consumer's input size: whenever `producer` has generated n values written to location x at different times, we have $\text{TRMS}_{\texttt{consumer}} = n$. Indeed, all read operations on x are induced first-accesses: thanks to the interleaving guaranteed by semaphores, each `read`(x) in `consumeData` is always preceded by a `write`(x) in `produceData`.

procedure `externalRead()`

```
1: let b a buffer of size 2
2: for i = 1 to n do
3:     load b with external data    // does not imply read of b
4:     consumeData(b[0])            // read and process b[0]
```

Figure 3. Buffered read from an external device: after n iterations, $\text{RMS}_{\texttt{externalRead}} = 1$ and $\text{TRMS}_{\texttt{externalRead}} = n$.

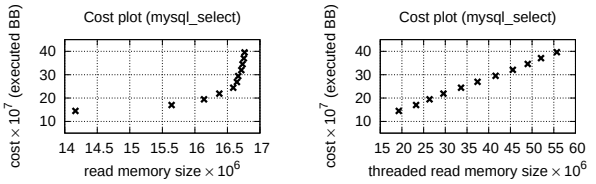
Example 4. The example in Figure 3 describes the case of buffered read operations. Procedure `externalRead` loads $2n$ values from an external device (line 3): this is done by the operating system that fills in buffer b with new data at each iteration. These load operations, however, should not be implicitly regarded as read operations performed by the running thread: as shown in line 4, only one of the two values loaded at each iteration is actually read and processed by procedure `externalRead`. Hence, at the end of the execution $\text{TRMS}_{\texttt{externalRead}} = n$, due to the n induced first-accesses at line 4. Notice that $\text{RMS}_{\texttt{externalRead}} = 1$ since data items are loaded across iterations on the same two memory locations $b[0]$ and $b[1]$ but only $b[0]$ is repeatedly read. We will further discuss the interaction between kernel system calls and threads in Section 4.3.

3. Case Studies

In this section we discuss the utility of the TRMS metric in real applications. We show a variety of cases where TRMS correctly characterizes the input size where RMS either fails or does not collect sufficient profiling data. Our examples are based on the `aprof-trms` tool described in Section 5 and use basic block (BB) counts as performance metric.

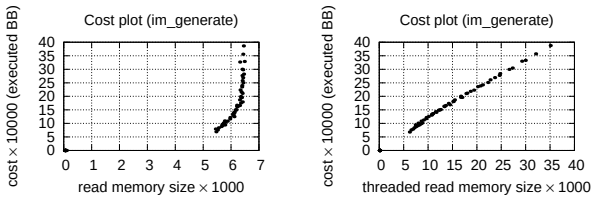
Input sensitive profiles can be naturally used to produce performance charts where some cost measure is plotted against the TRMS or the RMS. For instance, for each distinct input size n of a routine r, we can plot the maximum time spent by an activation of r on input size n (worst-case running time plot) or the number of times r was activated on an input of size n (workload plot). Similar charts could be produced for different cost measures (e.g., average running time), though we will not use them throughout this section.

Our discussion is based on two different applications: `MySQL`, a relational database management system [21], and `vips`, an image processing software package included in



(a) (b)

Figure 4. Function `mysql_select` of `MySQL`: worst-case running time plots respectively obtained using RMS or TRMS as an estimate for the input size.

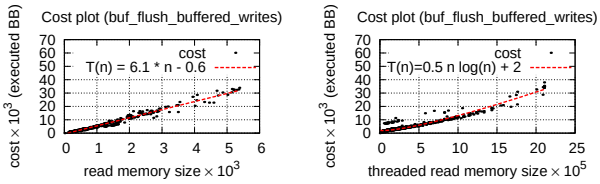


(a) (b)

Figure 5. Function `im_generate` of `vips` (PARSEC 2.1): worst-case running time plots respectively obtained using RMS or TRMS as an estimate for the input size.

the PARSEC 2.1 benchmark suite [3]. `MySQL` manages every new connection to the database by means of a separate thread, which contends for access to different shared data structures, and uses both I/O and network intensively. We also remark that `vips` is a data-parallel application, which constructs multi-threaded image processing pipelines in order to apply fundamental image operations such as affine transformations and convolutions.

Impact of input size estimation on asymptotic trends. If the input size is not estimated correctly, the analysis of profiling data can lead to misleading results. Consider for instance the following scenario: we have n activations $r_1 \dots r_n$ of a routine r, activation r_i has cost i and performs i read operations, out of which $\lceil i/2 \rceil$ are first-accesses and $\lfloor i/2 \rfloor$ are induced first-accesses. Hence, $\text{TRMS}_{r_i} = \lceil i/2 \rceil + \lfloor i/2 \rfloor = i$ while $\text{RMS}_{r_i} = \lceil i/2 \rceil$. Notice that $\text{TRMS}_{r_i} \geq \text{RMS}_{r_i}$, in accordance with Inequality 1. In the worst-case running time plot obtained using the TRMS we have n distinct points and the running time grows as the function $f(x) = x$. Conversely, in the worst-case running time plot obtained using the RMS we have only $n/2$ points: any two consecutive activations r_i and r_{i+1} (with i odd) have the same RMS value $\lceil i/2 \rceil$ and the worst-case cost is $i + 1$ (i.e., the maximum between costs i and $i + 1$ of the two activations). Hence, the running time appears to grow as the function $f(x) = 2x$. The problem would be even more critical if, e.g., $\text{RMS}_{r_i} = \lfloor \log i \rfloor$: in this case, in the worst-case plot obtained using the RMS, the running time would appear to grow exponentially as $f(x) = 2^x$.



(a) (b)

Figure 6. Function `buf_flush_buffered_writes` of `MySQL`: worst-case running time plots with curve fitting.

As shown by figures 4, 5, and 6, similar phenomena can arise in practice in I/O bounded or multithreaded applications. The running time of routine `mysql_select` in Figure 4 appears to grow (at least) quadratically when we measure the input size by means of the RMS, and linearly using the TRMS. In this experiment, the query operation simply selects all tuples in the table and is applied to tables of increasing sizes: at each query, tuples are partitioned into groups, each group is loaded into a buffer through a kernel system call and is then read by routine `mysql_select`. The RMS does not count repeated buffer read operations: hence, the input size on larger tables is exactly the same as in smaller ones (it roughly coincides with the buffer size), while the running time grows due to the larger number of buffer loads.

Routine `im_generate` in benchmark `vips` shows an analogous effect (see Figure 5). In this case the induced first-accesses not counted in the RMS are due to the interaction between threads via shared memory. In both examples, the RMS plot appears to reveal an asymptotic bottleneck, which instead does not actually exist. In other cases, the scenario might be the opposite: the RMS may not reveal the existence of a possible performance bottleneck, which can be instead characterized using the TRMS. For instance, the TRMS plot of routine `buf_flush_buffered_writes` of `MySQL` in Figure 6 shows a superlinear running time, while the RMS plot only suggests a linear growth, as highlighted by standard curve fitting techniques.

Profile richness. The usefulness of input-sensitive profile data crucially depends on the number of distinct input size values collected for each routine: each value corresponds to a point in the cost plots associated to a routine, and plots with a small number of points do not clearly expose the behavior of the routine. In our experiments, we observed that the use of TRMS instead of RMS can often yield a larger number of distinct input size values and thus more informative plots.

An example is provided by Figure 7: while routine `wbuffer_write_thread` was called 110 times during the execution of application `vips`, according to the RMS metric all its input sizes collapsed onto two distinct values (67 and 69, as shown in Figure 7a). However, this routine performs many load operations from secondary memory: hence, if we also take into account external input (Figure 7b), or external input combined with thread-input (Figure 7c), the number

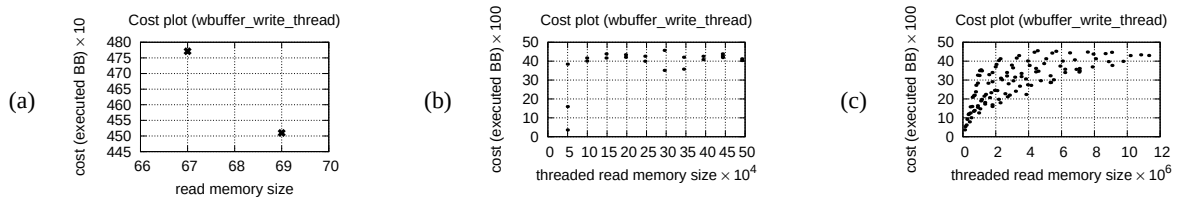


Figure 7. Function `wbuffer_write_thread` of `vips` (PARSEC 2.1): (a) RMS cost plot; (b) TRMS cost plot with external input only; (c) TRMS cost plot with both external and thread input.

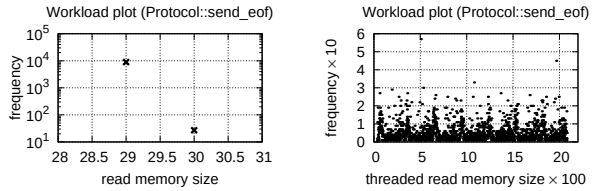


Figure 8. Function `Protocal::send_eof` of `MySQL`: workload plots respectively obtained using RMS or TRMS as an estimate for the input size.

of distinct TRMS values grows considerably and the trend in the cost plots becomes more meaningful.

Workload and input characterization. An additional benefit of input-sensitive profiling is the capability of characterizing the typical workloads on which a routine is called in the context of deployed systems. Richer profile data collected using the TRMS metric yield more accurate workload characterization, as shown, e.g., by the workload plots of Figure 8. Moreover, our multithreaded input-sensitive profiling methodology can also provide insights on the amount of interaction of each routine with external devices (external input) and cooperating threads (thread-induced input): for instance, 99.9% of the input of routine `wbuffer_write_thread` (Figure 7) is due to loads from secondary memory and thread interaction.

For each routine, we can automatically distinguish between external and thread-induced input. If we sort in decreasing order all routines in accordance with their percentage of induced first-accesses, we obtain an interesting characterization of the interplay between workload, computation, and concurrency, as shown in Figure 9. This figure plots, for each routine of benchmarks `MySQL` and `vips`, the percentage of induced first-accesses partitioned between external input and thread-induced input: a first look reveals that induced first-accesses of the majority of `MySQL` routines are due to external input, differently from `vips` routines where thread input is predominant. We remark that charts of this kind can be automatically produced by our profiler. In Section 6 we will provide a quantitative evaluation of profile richness and input characterization in a variety of applications on typical workloads.

4. Computing the Multithreaded Input Size

In this section we describe an efficient algorithm for computing the threaded read memory size of a routine activation and the input-sensitive profile of a routine. Routine profiles are thread-sensitive, i.e., profiles generated by routine activations made by different threads are kept distinct (if necessary, they can be combined in a subsequent step).

The profiler is given as input multiple traces of program operations associated with timing information. Each trace is generated by a different thread and includes: routine activations (`call`), routine completions (`return`), `read`/`write` memory accesses, and read/write operations performed through kernel system calls (`kernelRead` and `kernelWrite`, necessary to characterize external input).

As a first step, thread-specific traces are logically merged, interleaving operations performed by different threads according to their timestamps, in order to produce a unique execution trace. If two or more operations issued by different threads have the same timestamp, ties are broken arbitrarily: no assumption can be therefore done about which operation will be processed first. We remark that after merge and tie breaking, trace events are totally ordered. For simplicity of exposition, we also assume that `switchThread` events are inserted in the merged trace between any two operations performed by different threads.

For each operation issued by a routine r in a thread t, the profiler must update TRMS and cost information of r with respect to t. Some operations might also require to update profiling data structures related to threads other than t. To clarify the relationships between different threads, we first discuss a naive approach as a warm-up for the reader.

4.1 Naive Approach

Let t be a thread and let r be a routine activated by t. With a slight abuse of notation, we will denote with $\text{TRMS}_{r,t}$ the threaded memory size of a specific activation of r in t. According to the definition of multithreaded input size (see Section 2), computing $\text{TRMS}_{r,t}$ requires to count read operations issued by routine r that are either first accesses or induced first-accesses. In turn, identifying induced first-accesses requires to monitor write operations performed by *all* threads, i.e., performed also by threads different from t.

A simple-minded approach, which is sketched in Figure 10, is to maintain a set $L_{r,t}$ of memory locations ac-

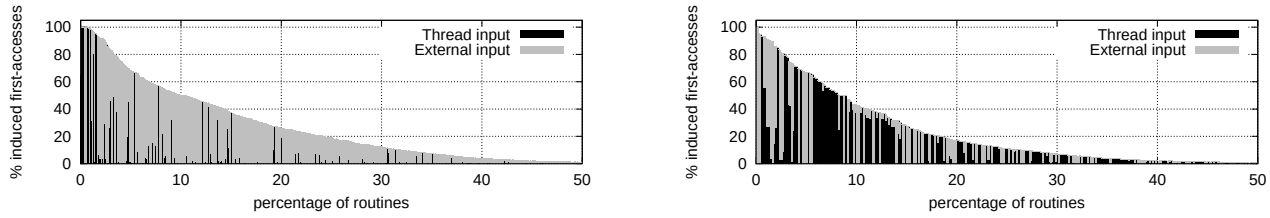


Figure 9. Thread-induced vs. external input on benchmarks (a) `MySQL` and (b) `vips`.

$\texttt{read}_t(\ell)$	if $\ell \notin L_{r,t}$ then $\text{TRMS}_{r,t}$++ $L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{write}_t(\ell)$	$L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{read}_{t'}(\ell)$, $t' \neq t$	–
$\texttt{write}_{t'}(\ell)$, $t' \neq t$	$L_{r,t} \leftarrow L_{r,t} \setminus \{\ell\}$

Figure 10. Computation of $\text{TRMS}_{r,t}$ with a naive approach. The notation $\texttt{read}_t/\texttt{write}_t(\ell)$indicates that location ℓ is read/written by thread t.

cessed during the activation of r. Immediately after entering r, this set is empty and $\text{TRMS}_{r,t} = 0$. Memory locations can be both added to and removed from $L_{r,t}$ during the execution of r, as follows:

- when r reads or writes a location ℓ, then ℓ is added to $L_{r,t}$ (if not already present);
- when a thread $t' \neq t$ writes a location ℓ, then ℓ is removed from $L_{r,t}$ (if present): this allows it to recognize induced first-accesses.

With this approach, at any time during the execution of r, a read operation on a location ℓ is a first access (possibly induced by other threads) if and only if $\ell \notin L_{r,t}$. Hence, $\text{TRMS}_{r,t}$ is increased only if this test succeeds. Notice that read operations performed by threads different from t change neither set $L_{r,t}$ nor $\text{TRMS}_{r,t}$.

We remark that in the description above r can be any routine in the call stack of thread t (not necessarily the topmost). Hence, the same checks and updates must be performed for all pending routine activations in the call stack of t. Due to stack-walking and to the fact that write operations require to update sets $L_{r,t}$ of *all* threads, this simple-minded approach is extremely time-consuming. It is also very space demanding: in the worst case, each distinct memory location could be stored in all sets $L_{r,t}$ for each thread t and each routine activation r pending in the call stack of t. In that case, the space would be proportional to the memory size times the maximum stack depth times the number of threads.

4.2 The Read/Write Timestamping Algorithm

To obtain a more space- and time-efficient algorithm, we exploit the latest-access approach described in [5]. Namely, we avoid to store explicitly the threaded read memory size $\text{TRMS}_{r,t}$ and the sets $L_{r,t}$ of accessed memory locations. Instead, we maintain partial information that can be updated quickly during the computation and from which the TRMS can be easily derived upon the termination of a routine.

In more details, we adapt the latest-access approach [5] as follows: for each thread t and memory location ℓ, we store ℓ in only one set $P_{r,t}$ such that r is the latest routine activation in t that accessed ℓ (either directly or by its completed subroutines). At any time during the execution of thread t and for each pending routine activation r, it holds:

$$L_{r,t} = P_{r,t} \cup \{P_{r',t} : r' \; descendant \; of \; r\}$$

where r' is any pending routine activation that is above r in the call stack at that time. Sets $P_{r,t}$ will be stored implicitly by associating timestamps to routines and memory locations.

Similarly to the naive approach of Figure 10, locations will be both added to and removed from $P_{r,t}$ to characterize induced first-accesses. However, this turns out to be inefficient in a multithreaded scenario: differently from read operations that change only thread-specific sets, write accesses require to change the sets $P_{r,t}$ of each activation r pending in the call stack of each running thread t. By implicitly updating only one set $P_{r,t}$ per thread, the latest-access algorithm avoids stack walking, but the update time for write accesses is still linear in the number of threads, which can be prohibitive in practice.

To overcome this problem, we combine the latest access approach with global timestamps that are appropriately updated upon write accesses to memory locations: in this way, we will recognize induced first-accesses by comparing thread-specific timestamps with global ones. The entire algorithm is sketched in Figure 11.

Data structures. The algorithm uses the following global data structures:

- a counter $count$ that maintains the total number of thread switches and routine activations across all threads;
- a shadow memory wts such that, for each memory location ℓ, $wts[\ell]$ is the timestamp of the latest write operation on ℓ performed by any thread. The timestamp of a memory access is defined as the value of $count$ at the time in which the access took place.

procedure `call`(r, t)
1: $count$ ++
2: top_t ++
3: $S_t[top_t].rtn \leftarrow r$
4: $S_t[top_t].ts \leftarrow count$
5: $S_t[top_t].trms \leftarrow 0$
6: $S_t[top_t].cost \leftarrow$ `getCost()`

procedure `return`(t)
1: `collect(`$S_t[top_t].rtn$, $S_t[top_t].trms$, `getCost()-`$S_t[top_t].cost$`)`
2: $S_t[top_t\text{-}1].trms$ += $S_t[top_t].trms$
3: top_t--

procedure `switchThread()`
1: $count$ ++

procedure `read`(ℓ, t)
1: **if** $ts_t[\ell] < wts[\ell]$ **then**
2: $S_t[top_t].trms$ ++
3: **else**
4: **if** $ts_t[\ell] < S_t[top_t].ts$ **then**
5: $S_t[top_t].trms$ ++
6: **if** $ts_t[\ell] \neq 0$ **then**
7: $i =$ max idx s.t. $S_t[i].ts \leq ts_t[\ell]$
8: $S_t[i].trms$ --
9: **end if**
10: **end if**
11: **end if**
12: $ts_t[\ell] \leftarrow count$

procedure `write`(ℓ, t)
1: $ts_t[\ell] \leftarrow count$
2: $wts[\ell] \leftarrow count$

Figure 11. TRMS algorithm: multi-threaded input.

Similarly to [5], the algorithm also uses the following thread-specific data structures for each thread t:

- a shadow memory ts_t such that, for each memory location ℓ, $ts_t[\ell]$ is the timestamp of the latest access (read or write) to ℓ made by thread t;
- a shadow run-time stack S_t, whose top is indexed by variable top_t. For each $i \in [1, top_t]$, the i-th stack entry $S_t[i]$ stores:
  - The id $S_t[i].rtn$, the timestamp $S_t[i].ts$, and the cumulative cost $S_t[i].cost$ of the i-th pending routine activation.
  - The *partial threaded read memory size* $S_t[i].trms$ of the activation, defined so that the following invariant property holds throughout the execution for each i such that $1 \leq i \leq top_t$:

$$\forall i,\ 1 \leq i \leq top_t : \quad \text{TRMS}_{i,t} = \sum_{j=i}^{top_t} S_t[j].trms \qquad (2)$$

  where $\text{TRMS}_{i,t}$ is a shortcut for $\text{TRMS}_{S_t[i].rtn,t}$. At any time, $\text{TRMS}_{i,t}$ equals the current TRMS value of the i-th pending activation on the portion of the execution trace generated by thread t seen so far.

As shown in [5], Invariant 2 implies the following interesting property: for each pending routine activation, its TRMS value can be obtained by summing up its partial threaded read memory size with the TRMS value of its (unique) pending child, if any. More formally:

$$\text{TRMS}_{top_t,t} = S_t[top_t].trms$$

$$\text{TRMS}_{i,t} = S_t[i].trms + \text{TRMS}_{i+1,t}$$

for each $i \in [1, top_t - 1]$. Hence, if we can correctly maintain the partial threaded read memory size during the execution, upon completion of a routine we will also get the correct TRMS value.

Algorithm and analysis. The partial threaded read memory size can be maintained as shown in Figure 11. We first notice that the global timestamp counter $count$ is increased at each thread switch and routine call, and its value is used to update routine timestamps (line 4 of procedure `call`), global memory timestamps (line 2 of procedure `write`), and local memory timestamps (lines 1 and 12 of procedures `write` and `read`, respectively). Upon activation of a routine, procedure `call`(r, t) creates and initializes a new shadow stack entry for routine r in S_t. When the routine activation terminates, its cost is collected and its partial TRMS (which at this point coincides with the true TRMS value according to equation $\text{TRMS}_{top_t,t} = S_t[top_t].trms$ discussed above) is added to the partial TRMS of its parent, preserving Invariant 2.

Local timestamps of memory locations are updated both by read and write accesses, while global timestamps are not updated upon read operations (they are thus associated to write operations only). This update scheme makes it possible to recognize induced first-accesses to any location ℓ, which is done by lines 1-2 of procedure `read`. If the read-/write timestamp $ts_t[\ell]$ local to thread t is smaller than the global write timestamp $wts[\ell]$, then location ℓ must have been written more recently than the last read/write access to ℓ by thread t. Note that, if the latest access to ℓ was a write operation by thread t, then it would be $ts_t[\ell] = wts[\ell]$ (see procedure `write`), letting the test $ts_t[\ell] < wts[\ell]$ fail. Hence, if the test succeeds, the last write on ℓ must have been done by some thread $t' \neq t$, the read access by t is an induced access, and the partial TRMS of the topmost routine is correctly increased by line 2 of procedure `read`. Invariant 2 is fully preserved by this assignment: the accessed value is new not only for the topmost routine in the call stack S_t, but also for all its ancestors, whose TRMS is implicitly updated according to Equation 2.

On the other side, if the test of line 1 of procedure `read` fails, the read access to ℓ might still be a first access: this happens if the last access to location ℓ by thread t took place before entering the current (topmost) routine. Lines 4–10 address this case, updating the partial TRMS as described in [5]: the partial TRMS of the topmost routine is increased, while the partial TRMS of an appropriately chosen ancestor is decreased (it is proved in [5] that Invariant 2 is preserved).

The running time of all operations is constant, except for line 7 of procedure `read` that requires $O(\log d_t)$ worst case time, where d_t is the depth of the call stack S_t.

4.3 External Input

In Section 4.2 we have focused on induced first-accesses generated by multi-threaded executions. In this section we show that the read/write timestamping algorithm can be nat-

procedure `kernelWrite`(ℓ)
1: $count$ ++
2: $wts[\ell] \leftarrow count$

procedure `kernelRead`(ℓ, t)
1: `read`(ℓ, t)

Figure 12. TRMS algorithm: external input.

urally extended to take into account also induced accesses due to external input.

Procedures `kernelRead` and `kernelWrite` shown in Figure 12 update the profiler's data structures when memory accesses are mediated by kernel system calls. Threads invoke system calls to get data from external devices (e.g., a disk or the network) or to send data to external devices. We remark that the operating system kernel must be treated differently from normal threads in our algorithm, since there are no kernel-specific shadow memory and shadow stack.

When a thread sends data to an external device, it must delegate the operating system to read the memory locations containing those data and write their content to the device. Hence, a thread external write operation corresponds to a `kernelRead` event in the execution trace. As shown in Figure 12, read memory accesses by the operating system are regarded as read operations implicitly performed by the thread, as if the system call were a normal subroutine. Upon a `kernelRead` event, it is therefore sufficient to invoke procedure `read` that, if necessary, will update the threaded TRMS of pending routine activations.

The case of `kernelWrite` operations is slightly more subtle. When a thread needs data from a external device, it delegates the operating system to write the device data to some memory buffer (if the buffer consists of n memory locations, the execution trace will contain n distinct `kernelWrite` events). This buffer load, however, should not be regarded as a thread external read operation: indeed, it may happen that only a subset of the loaded memory locations will be actually processed (and thus read) by the thread, and only those subset should be counted as external input. For this reason procedure `kernelWrite` does not directly change the partial TRMS of the topmost routine. Instead, it first increases $count$ and then associates buffer memory locations with a global write timestamp that is larger than any thread-specific timestamp. This forces the test $ts_t[\ell] < wts[\ell]$ to succeed if a buffer location ℓ will be subsequently read by the thread, properly increasing the partial TRMS only for actual read operations.

4.4 Counter Overflows

The global counter used by the read/write timestamping algorithm is common to all running threads and in our initial experiments was affected by frequent overflows, especially for long-running applications. Unfortunately, overflows are a serious concern in the computation of the TRMS, since they alter the partial ordering between memory timestamps yielding wrong input size values. To overcome this issue, we perform a periodical global renumbering of timestamps in the profiler's data structures, taking care not to alter the partial order between $ts_t[\ell]$, $wts[\ell]$, and $S_t[i].ts$ for each memory location ℓ, running thread t, and $1 \leq i \leq top_t$. Instead, we exploit the following observation: in Figure 11 there is no comparison between $wts[\ell]$ and $wts[\ell']$ or between $ts_t[\ell]$ and $ts_t[\ell']$, for $\ell \neq \ell'$. Hence, the order between timestamps of different memory locations is irrelevant and can be arbitrarily changed.

procedure `counterOverflow()`
1: **for each** running thread t **do**
2: **for** $i = 1$ **to** top_t **do**
3: add $S_t[i].ts$ to set A of active timestamps
4: `sort`(A)
5: **for each** running thread t **do**
6: **for** $i = 1$ **to** top_t **do**
7: $p =$ position of timestamp $S_t[i].ts$ in A
8: $S_t[i].ts = 3 \cdot p$
9: **for each** memory location ℓ **do**
10: $q =$ max idx in A s.t. $wts[\ell] \leq A[q]$
11: **for each** running thread t **do**
12: **if** $ts_t[\ell] < A[q] \vee ts_t[\ell] \geq A[q+1]$ **then**
13: $j =$ max idx in A s.t. $ts_t[\ell] \geq A[j]$
14: $ts_t[\ell] = 3 \cdot j$
15: **elif** $wts[\ell] = ts_t[\ell]$ **then** $ts_t[\ell] = 3 \cdot q + 1$
16: **elif** $wts[\ell] > ts_t[\ell]$ **then** $ts_t[\ell] = 3 \cdot q$
17: **else** $ts_t[\ell] = 3 \cdot q + 2$
18: $wts[\ell] = 3 \cdot q + 1$
19: $count = 3 \cdot |A| + 3$

Figure 13. Counter overflow procedure.

Our renumbering algorithm is sketched in Figure 13. For the sake of efficiency, the algorithm checks and renumbers each timestamp only once. Lines 1-4 collect all timestamps in the call stacks of running threads and sort them in increasing order. Notice that these timestamps are distinct: $count$ is increased by procedure `call` (see Figure 11) so that a new activation is always assigned an unused value, and the renumbering algorithm keeps the property true.

Routine timestamps are re-assigned in lines 5-8: the new timestamps are multiples of 3 (this choice will be justified below) and are chosen according to the rank of the original routine timestamp in the sorted set A. This guarantees that the original ordering between any two routine timestamps is preserved, and that the maximum value of a timestamp will be proportional to the total number of pending routine activations (i.e., $|A|$).

Thread-specific and global timestamps of memory locations are re-assigned in lines 9-18. According to line 10, let $A[q]$ be the latest pending routine activation (in any thread) started before the latest write to memory location ℓ. A thread t could have accessed ℓ before this activation (i.e., $ts_t[\ell] < A[q]$), between pending routine activations $A[q]$ and $A[q+1]$

(i.e., $A[q] \leq ts_t[\ell] < A[q+1]$), or after pending routine activation $A[q+1]$ (i.e., $ts_t[\ell] \geq A[q+1]$). If $q+1$ is not a valid index for A and $ts_t[\ell] \geq A[q]$ we can assume to be in the second case. The first and the third cases can be treated by assigning $ts_t[\ell]$ with the same value used for the most recent activation j such that $ts_t[\ell] \geq A[j]$ (lines 12-14): this guarantees that comparisons between $ts_t[\ell]$ and any routine timestamp at lines 4 and 7 of procedure `read` will succeed if and only if they succeeded before renumbering. The second case ($A[q] \leq ts_t[\ell] < A[q+1]$) requires to distinguish between three different situations, which explains why new routine timestamps are chosen in line 8 as multiples of 3:

1. $wts[\ell] = ts_t[\ell]$: t was the last thread to write location ℓ. After renumbering (lines 15 and 18), $ts_t[\ell] = wts[\ell] = 3q+1$. This guarantees that both $A[q] = 3q \leq ts_t[\ell] = wts[\ell] = 3q+1 < A[q+1] = 3(q+1)$;
2. $wts[\ell] > ts_t[\ell]$: thread t has accessed location ℓ before its last write. In this case the new timestamp of $ts_t[\ell]$ is $3q$ (line 16). This preserves both the relations $ts_t[\ell] = 3q < wts[\ell] = 3q+1$ and $ts_t[\ell] = 3q < A[q+1] = 3(q+1)$;
3. $ts_t[\ell] > wts[\ell]$: thread t has read location ℓ after its last write. In this case $ts_t[\ell]$ gets the new value $3q+2$ (line 17). The order relation $wts[\ell] = 3q+1 < ts_t[\ell] = 3q+2 < A[q+1] = 3(q+1)$ remains valid.

Notice that the global timestamp $count$ is assigned with a value larger than all the other timestamps (line 19).

Using binary search to implement lines 7, 10, and 13, the running time of the global renumbering algorithm is $O(\rho \log \rho + \mu \tau \log \rho)$ where τ, μ, and ρ are the numbers of running threads, distinct memory cells, and pending routine activations, respectively. This can be amortized against $\Omega(2^w)$ thread switch and routine call operations, where w is the word memory size.

5. Implementation

To prove the feasibility of our approach, we implemented a threaded input-sensitive profiler by developing a Valgrind [23] tool called `aprof-trms`. Valgrind provides a dynamic instrumentation infrastructure that translates the binary code into an architecture-neutral intermediate representation (VEX). Analysis tools provide callbacks for events generated by the stream of VEX executed instructions.

Instrumentation. Similarly to the input-sensitive profiler described in [5], our tool traces all memory accesses and function calls and returns. We count basic blocks as a performance measure: tracing function calls and returns requires to instrument each basic block, thus counting basic blocks adds a light burden to the analysis time overhead, and improves accuracy in characterizing asymptotic behavior even on small workloads. Measuring basic blocks rather than time has several other advantages, very well motivated in [8]. In order to take into account external input, system calls are wrapped and properly mapped to one or more `kernelRead` or `kernelWrite` events: among the main system calls on a Linux x86_64 machine, `write`, `sendto`, `pwrite64`, `writev`, `msgsnd`, and `pwritev` correspond to `kernelRead` events, while `read`, `recvfrom`, `pread64`, `readv`, `msgrcv`, and `preadv` correspond to `kernelWrite` events.

Thread interleaving. Under Valgrind, threads of a traced application are serialized. This makes the development and debugging of a dynamic analysis framework and of its derived tools easier. Serialization should not be seen as a crucial limitation of our implementation: for instance Helgrind [19, 28] and DRD [28], two popular tools for detecting synchronization errors in programs that use the POSIX `pthreads` primitives, are both based on Valgrind. Serialization implies that our profiler does not need to perform tie breaking of events (see Section 4). However, in a serialized scenario the scheduling of threads becomes a critical concern: thread interleaving may be altered and the execution may deviate from non-serialized executions. In order to avoid unrealistic executions, our tool takes benefit of the fair thread scheduler introduced in the latest release of Valgrind.

Shadow memories. To reduce space overhead in practice, we maintain global and thread-specific shadow memories using three-levels lookup tables. A similar approach is also adopted by other prominent tools, such as `memcheck` [25]. A primary table indexes 2048 secondary tables, each covering 1GB of address space by indexing 16K chunks. Each chunk, in turn, contains the set of 32-bit timestamps for 64KB address space. In this way only chunks related to memory cells actually accessed by a thread need to be shadowed in its thread-specific shadow memory. Hence, on average (e.g., with embarrassingly parallel applications), the accessed primary memory is roughly partitioned among all running threads: the overhead for maintaining global and thread-specific shadow memories is therefore considerably smaller than in the worst-case scenario (where it would be proportional to number of running threads $\times$ number of distinct memory cells). Experiments in Section 6 will largely confirm this hypothesis.

6. Experimental Evaluation

In this section we discuss the results of an extensive experimental evaluation of `aprof-trms` on a variety of benchmarks, including the SPEC OMP2012 [20] and the Princeton Application Repository for Shared-Memory Computers (PARSEC 2.1) [3]. The goals of our experiments are threefold: studying the slowdown and space overhead of `aprof-trms` compared to other heavyweight dynamic analysis tools, evaluating the benefits of the TRMS with respect to the RMS, and characterizing the amount of thread-induced and external input on the considered benchmarks.

	TIME							SPACE						
	secs	slowdown						MB	overhead					
	native	nul grind	mem check	call grind	hel grind	aprof rms	aprof trms	native	nul grind	mem check	call grind	hel grind	aprof rms	aprof trms
350.md	3184.5	6.0	43.5	34.7	125.4	39.6	41.2	50.8	1.9	2.0	2.0	6.7	2.2	2.3
351.bwaves	192.0	22.1	–	68.4	92.0	78.0	91.2	1582.1	1.1	–	1.1	4.0	3.0	4.0
352.nab	185.0	21.1	111.4	80.4	127.2	107.5	186.7	57.0	1.7	2.4	1.8	5.8	2.5	2.8
358.botsalgn	8.0	42.8	85.6	132.9	114.2	146.3	179.3	57.7	1.6	1.7	1.7	3.4	1.8	2.3
359.botsspar	1.0	204.0	353.9	523.1	368.3	301.4	457.6	62.9	1.6	2.1	1.7	4.8	2.4	2.6
360.ilbdc	1936.8	2.2	18.5	4.7	64.9	17.3	26.2	1415.2	1.0	1.1	1.0	1.8	4.1	5.1
362.fma3d	46.6	22.7	113.4	45.2	393.4	99.0	118.0	155.8	1.3	2.5	1.3	10.6	3.0	3.8
367.imagick	170.0	24.1	91.0	50.5	141.6	52.3	60.6	77.9	1.6	2.0	1.7	5.4	3.1	3.9
370.mgrid331	5.2	39.7	101.7	50.9	194.6	95.7	130.2	395.1	1.1	1.2	1.1	1.8	2.1	3.0
371.applu331	26.6	45.0	230.2	103.7	472.8	228.9	367.6	88.2	1.5	1.8	1.6	5.6	3.2	3.7
372.smithwa	14.4	28.8	78.4	57.0	148.1	167.8	213.9	49.4	1.8	1.9	2.0	3.3	2.4	2.6
376.kdtree	33.9	19.5	99.4	127.3	366.6	247.4	551.0	98.6	1.4	2.8	1.5	8.4	4.4	5.6
geometric mean		23.6	94.1	64.8	179.4	101.5	140.8		1.4	2.0	1.5	4.5	2.8	3.3

Table 1. Performance comparison of `aprof-trms` with `aprof` and some prominent Valgrind tools on the SPEC OMP2012 benchmarks.

6.1 Experimental Setup

Benchmarks. The OMP2012 benchmark suite of the Standard Performance Evaluation Corporation [20] is a collection of fourteen OpenMP-based applications from different science domains. All of them were run on the SPEC input `train` workloads in 64-bit mode. We could successfully test all the components except for `bt331` and `swim`, whose execution failed due to a Valgrind memory issue.

The Princeton Application Repository for Shared-Memory Computers (PARSEC 2.1) is a benchmark suite for studies of Chip-Multiprocessors [3]. It includes different workloads chosen from a variety of areas such as computer vision, media processing, computational finance, enterprise servers, and animation physics. PARSEC defines six input sets for each benchmark: experimental results reported in this section are all based on the `simlarge` sets [3].

For the sake of completeness, we also included in our tests the `MySQL` application (version 5.5.30) discussed in Section 3: we used the `mysqlslap` load emulation client [22], simulating 50 concurrent clients that submit approximately 1000 auto-generated queries.

Metrics. Besides slowdown and space overhead, we use the following metrics:

1. *Routine profile richness*: for each routine r, let $|\text{RMS}_r|$ be the number of distinct RMS values collected for routine r (each value corresponds to a point in the graphs associated with r). Similarly, let $|\text{TRMS}_r|$ be the number of distinct TRMS values collected for routine r for all threads. The profile richness of routine r is defined as:

$$\frac{|\text{TRMS}_r| - |\text{RMS}_r|}{|\text{RMS}_r|}$$

   Intuitively, this metric compares the number of distinct input values obtained using the TRMS and the RMS, respectively. We notice that $|\text{TRMS}_r| \geq |\text{RMS}_r|$ does not necessarily hold: it may happen that two distinct RMS values x and y (obtained from two different activations of a routine) correspond to the same TRMS value z, with $z \geq \max\{x, y\}$. Hence, the profile richness may be either positive, if more points are collected using the TRMS, or negative, if more points are collected using the RMS. We will see that in practice the latter case is seldom true.

2. *Input volume*: according to Inequality 1, the TRMS of a routine activation is always larger than or equal to the RMS of the same activation. The input volume metric characterizes the increase of the input size values due to multi-threading and to external input for an entire execution:

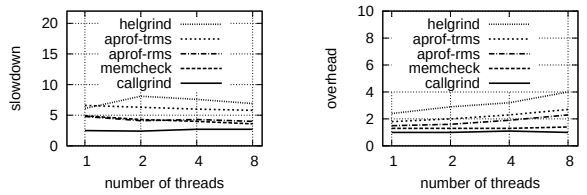
$$1 - \frac{\sum_{\text{routine activations } r} \text{RMS}_r}{\sum_{\text{routine activations } r} \text{TRMS}_r}$$

   Values of this metric range in $[0, 1)$. If $\text{TRMS}_r = \text{RMS}_r$ for all routine activations r, then the input volume is 0. Conversely, if $\text{TRMS}_r \gg \text{RMS}_r$ for all routine activations r, then the input volume gets close to 1.

3. *Thread-induced input*: this metric measures the percentage of induced first-accesses (line 2 of procedure `read` in Figure 11) due to multi-threading.

4. *External input*: similarly to the previous case, this metric measures the percentage of induced first-accesses due to external input.

Evaluated tools. We compared the performance of `aprof-trms` to four reference Valgrind tools: `nulgrind`, which does not collect any useful information and is used only for testing purposes, `memcheck` [25], a tool for detecting memory-related errors, `callgrind` [30], a call-graph generating profiler, and `helgrind` [19], a data race detector. Although the considered tools solve different analysis problems, all of them share the same instrumentation infrastructure provided by Valgrind, which accounts for a significant fraction of the execution times: `memcheck` does not trace function calls/returns and mainly relies on memory read-/write events; `callgrind` instruments function calls/returns, but not memory accesses, and `helgrind` analyzes concur-



(a) (b)

Figure 14. (a) Time and (b) space overhead, with respect to `nulgrind`, as a function of the number of threads.

rent applications. We also compared `aprof-trms` against a previous version of `aprof` based on the RMS metric (see Section 2): we remark that `aprof-rms` targets sequential computations only, without taking into account induced first-accesses.

Platform. Experiments were performed on a cluster machine with four nodes, each equipped with two 64-bit AMD Opteron Processors 6272 @ 2.10 GHz (32 cores), with 64 GB of RAM running Linux kernel 2.6.32 with `gcc` 4.4.7 and Valgrind 3.8.1 – SVN rev. 13126.

6.2 Experimental Results

Slowdown and space overhead. Performance figures of our evaluated tools on the SPEC OMP2012 benchmarks, obtained spawning four OpenMP threads per benchmark, are summarized in Table 1. We do not report results for the PARSEC benchmarks because some tools revealed invalid memory accesses and other memory issues that prevented a reliable comparison of executions under different tools.

Compared to native execution, all the evaluated tools exhibit a large slowdown: even `nulgrind`, which is reported to be roughly 5 times slower than native [28], in our experiments turned out to have a mean slowdown factor of $23.6\times$. `aprof-trms` is on average 6 times slower than `nulgrind`: this is worse than `memcheck`, which is 1.5 times faster than our tool but does not trace function calls and returns, and better than `helgrind`, which is 1.3 times slower than `aprof-trms` and is the only tool designed for the analysis of concurrent computations. Recognizing induced first-accesses causes a 38% overhead on the running time, as demonstrated by the comparison of `aprof-trms` with `aprof-rms`.

The mean memory requirements of `aprof-trms` are within a factor of 3.3 of native execution. This confirms our expectation: if memory is roughly partitioned among the four threads, the three-level lookup tables guarantee that the overall size of thread-specific shadow memories is proportional to the size of accessed memory locations. This should be added to the size of the global shadow memory, thus obtaining the total $3.3\times$ space overhead. Even tools that do not use shadowing, such as `nulgrind` and `callgrind`, require at least $1.4\times$ more space than native execution. `memcheck` hinges upon memory shadowing, but turns out to be more efficient than `aprof-trms` thanks to the adoption of memory compression schemes and to its independence from the number of threads. Similarly, `aprof-rms` is slightly more efficient than our tool due to the lack of a global shadow memory. On the other hand we remark that `helgrind`, which is akin to our tool with respect to the analysis of concurrency issues, uses 36% more space than `aprof-trms`.

Figure 14 shows the average slowdown and space overhead, with respect to the reference Valgrind tool `nulgrind`, as a function of the number of spawned OpenMP threads. All the evaluated tools appear to scale properly. The average slowdown slightly decreases as the number of threads increases: this is because Valgrind serializes the execution of threads, and the time spent for instrumentation can be better amortized over the serialized executions of a larger number of threads. Overall, this experiment confirms the results detailed in Table 1 in the case of four threads. The mean space overhead of `callgrind` and `memcheck` is roughly constant: their analyses are indeed independent from concurrency issues. On the other hand, `aprof-trms` and `helgrind` show a modest increase of the space overhead when the number of threads increases: our profiling of the memory usage of `aprof-trms` revealed that the space overhead mostly depends on shadow memories, whose total space usage, however, grows sublinearly with the number of threads. This confirms the effectiveness of our implementation based on three-level lookup tables. The comparison with `aprof-rms` also suggests that the space overhead due to the global shadow memory used by `aprof-trms` is better amortized as the number of threads increases.

TRMS versus RMS. Our second set of experiments aims at evaluating the benefits of the TRMS metric with respect to the RMS. As shown in [5], an RMS-based input-sensitive profiler can collect a significant number of distinct input values for most algorithmic-intensive functions, thus producing informative cost plots. A first natural question is whether using TRMS instead of RMS has any positive or negative impact on the profile richness. Charts in Figure 15 contribute to answer this question. Each curve is related to a specific benchmark. A point (x, y) on a curve means that $x\%$ of routines have profile richness at least y: e.g., in benchmark `dedup`, the number of points collected using the TRMS metric is more than 100 times larger than using the RMS for roughly 4% of the routines. As expected, only for a small percentage of routines $|\text{TRMS}_r|$ is much larger than $|\text{RMS}_r|$: this is due to the fact that I/O and thread communication are typically encapsulated in a small number of software components. However, for these routines $|\text{TRMS}_r|$ can be substantially larger than $|\text{RMS}_r|$, e.g., up to a factor of roughly 10^6 for benchmark `dedup`. We also notice that profile richness is negative only for a statistically intangible number of routines: this means that TRMS-based profiles are (almost) always at least as informative as those obtained using the RMS.

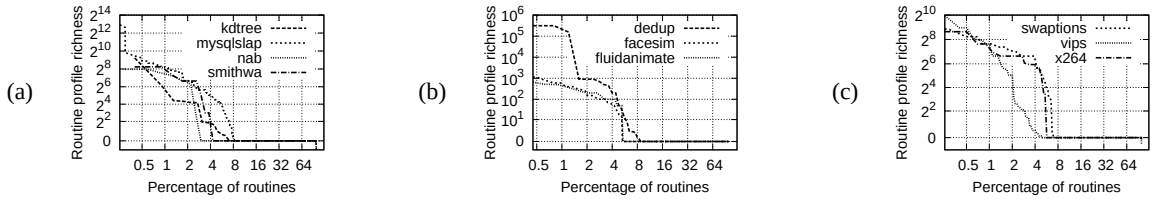


Figure 15. Routine profile richness of TRMS w.r.t. RMS for a representative set of benchmarks.

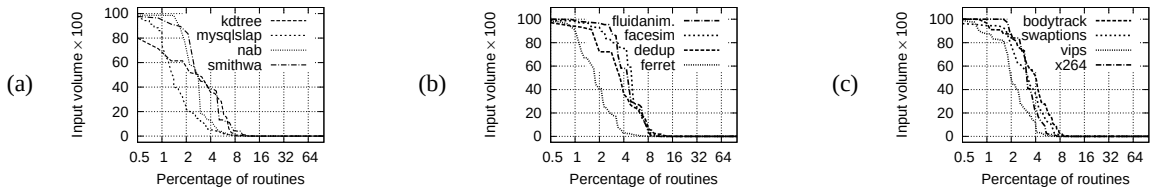


Figure 16. Input volume of TRMS w.r.t. RMS for a representative set of benchmarks.

Due to Inequality 1, TRMS values are always larger that RMS values for the same routine activations. Figure 16 characterizes the increase of the input size values due to induced first-accesses on a representative set of benchmarks. The interpretation of these graphs is similar to Figure 15: a point (x, y) on each benchmark-specific curve means that $x\%$ of routines have input volume $\geq y$. E.g., in benchmark `fluidanimate` roughly 3% of the routines take almost all their input from external devices or from other threads. The trend of curves in Figure 16 decreases steeply from 100 to 0, reaching its minimum at $x \simeq 8\%$ for most benchmarks: this means that 8% of the routines are responsible of thread intercommunication and streamed I/O, and the input size of these routines cannot be appropriately predicted by the RMS metric alone.

Analysis of induced first-accesses. In the previous experiments we observed that a non-negligible number of routines communicate with other threads or with the kernel via system calls. A natural question is how many induced first-accesses are due to external input or are thread-induced. Figure 17 answers this question, plotting the percentage of thread-induced and external input on a representative set of benchmarks: percentages are computed over the total number of induced first-accesses, and therefore sum up to 100%. Benchmarks are sorted by decreasing percentage of thread-induced input (and thus by increasing external input). An interesting observation is that the SPEC OMP2012 benchmarks get naturally clustered in the leftmost part of the histogram (from `nab` to `botsalgn`), and all of them have thread-induced input larger than 69%. We notice that external input is predominant in `vips`, which seems in contrast with Figure 9. This contradiction, however, is only apparent and has a clear explanation. Figure 9 plots external input on a routine-per-routine basis, while the global percentage in Figure 17 is routine-independent: the external input of a specific routine also includes the external input of all its descendants in the call tree, which is instead neglected in the global benchmark measure (where each induced first-access is counted only once in the percentage computation). Similar considerations apply to `mysqlslap`.

For the sake of completeness, Figure 18 and Figure 19 provide a quantitative evaluation of thread-induced and external input on a routine-per-routine basis: a point (x, y) on each benchmark-specific curve means that $x\%$ of routines have external / thread-induced input $\geq y\%$. For instance, in benchmark `dedup`, 16% of the routines are such that at least 20% of their induced first-accesses are due to thread intercommunication. These charts are in the spirit of Figure 9, but exploit a more compact representation.

7. Related Work

There is a vast literature on performance profiling, both at the inter- and intra-procedural level: see, e.g., [1, 2, 4, 9–11, 29, 33] and the references therein. All these works aim at associating performance metrics to distinct paths traversed in the call graph or in the control flow graph during a program's execution. Input-sensitivity issues are instead explored in [5, 8, 15, 31]. Marin and Mellor-Crummey [15] consider the problem of understanding how an application's performance scales given different problem sizes, using data collected from multiple runs with determinable input parameters. Goldsmith, Aiken, and Wilkerson [8] also propose to run a program on workloads of different sizes, to measure the performance of its routines, and eventually to fit these observations to a model that predicts how the performance scales. The workload size of the program's routines, however, is not computed automatically. Algorithmic profiling by Zaparanuks and Hauswirth [31], besides identifying boundaries between different algorithms in a program, infers their computational cost, which is related to the in-

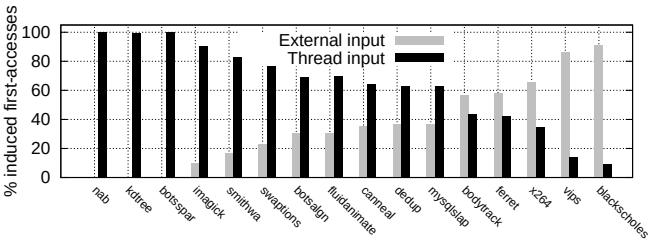


Figure 17. External vs. thread-induced input.

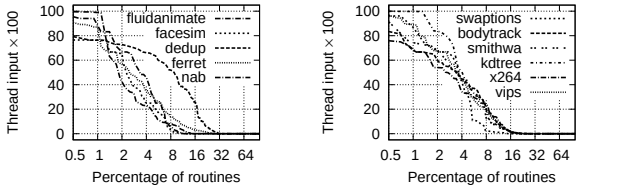


Figure 18. Thread-induced input on a routine basis.

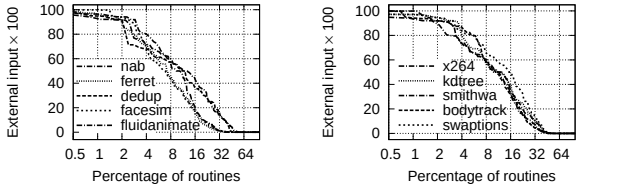


Figure 19. External input on a routine basis.

put size. The notion of input size is defined at a high level of abstraction, using different definitions for different data structures (e.g., the size of an array or the number of nodes in a tree). Instead, the input-sensitive profiling methodology described in [5], which provides the basis for our approach, automatically infers the input size by tracing low-level memory accesses performed by different routines. None of these approaches addresses concurrency issues, being thus limited to sequential computations.

The problem of empirically studying the asymptotic behavior of a program has been the target of extensive research in experimental algorithmics [6, 17, 18], where individual portions of algorithmic code are extracted from applications and separately analyzed on ad-hoc test harnesses. This approach has some drawbacks as a performance evaluation method in actual software development, most prominently the fact that, by studying performance-critical routines out of their context, possible performance effects due to the interaction with the overall application might be missed.

A variety of parallelism-related profilers have been proposed to help programmers parallelize complex codes by uncovering the dependencies between different regions of the program: examples include pp [14], Alchemist [32], and Kremlin [7]. Other profilers for multicore machines, such as [13, 16, 24], focus on NUMA-related performance issues and exploit detailed temporal information about memory accesses in order to build temporal flows of interactions between threads and objects. This is similar to our problem of relating memory accesses with thread intercommunication, although the final goal is orthogonal to ours, since these works aim at understanding the speed improvements that can result from parallelizing different portions of code, from executing a program on a parallel platform, or from diagnosing and reducing distant memory accesses.

8. Conclusions

In this paper we have extended the input-sensitive profiling methodology to concurrent computations. Input-sensitive profiling requires to measure automatically the size of the input given to a generic code fragment: in a multithreaded scenario, this raises a variety of interesting issues mainly due to thread intercommunication via shared memory. At this aim, we have proposed a novel metric, called TRMS, that gives an estimate of the size of the input of each routine activation by taking into account first-accesses, possibly induced by other threads or by kernel system calls. We have shown that our approach is both methodologically sound and practical. Namely, our Valgrind-based implementation achieves performances comparable to other prominent heavyweight analysis tools. As a future direction, it would be interesting to adapt our methodology to a fully scalable and concurrent dynamic instrumentation framework, in order to exploit parallelism to leverage the slowdown of our profiler.

Our methodology raises many interesting open issues regarding input characterization and thread intercommunication in concurrent applications. Measures derived from TRMS might allow it to evaluate concurrency-related aspects and to discover how multithreaded applications scale their work and how they communicate via shared memory. E.g., in a recent experimental study [12], it has been observed that even widespread multithreaded benchmarks do not interact much or interact only in limited ways, and that communication does not change predictably as a function of the number of cores. This study exploits a characterization of read/write memory accesses, and we believe that the TRMS might shed new light towards this direction.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997. ISSN 0362-1340.

[2] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, pages 168–179. ACM, 2001.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *17th Int. Conference on Parallel Architecture and Compilation Techniques (PACT'08)*, pages 72–81, 2008.

[4] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *CGO*, pages 205–216. IEEE Computer Society, 2005.

[5] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*, pages 89–98, 2012.

[6] C. Demetrescu, I. Finocchi, and G. F. Italiano. Algorithm engineering. *Bulletin of the EATCS (algorithmics column)*, 79:48–63, 2003.

[7] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *Proc. 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI'11)*, pages 458–469, 2011.

[8] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, pages 395–404, 2007.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler (with retrospective). In K. S. McKinley, editor, *Best of PLDI*, pages 49–57. ACM, 1982.

[10] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proc. Summer 1993 USENIX Technical Conference*, pages 1–19. USENIX Association, 1993.

[11] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[12] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *Proc. ACM int. conf. on Object oriented programming systems languages and applications (OOPSLA'12)*, pages 335–354, 2012.

[13] R. Lachaize, B. Lepers, and V. Quema. MemProf: a memory profiler for NUMA multicore systems. In *Proc. USENIX Annual Technical Conference (ATC'12)*, pages 53–64, 2012.

[14] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, 4(7):812–826, 1993.

[15] G. Marin and J. M. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proc. SIGMETRICS 2004*, pages 2–13, 2004.

[16] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS'10)*, pages 87–96, 2010.

[17] C. C. McGeoch. Experimental algorithmics. *Communications of the ACM*, 50(11):27–31, 2007.

[18] C. C. McGeoch, P. Sanders, R. Fleischer, P. R. Cohen, and D. Precup. Using finite experiments to study asymptotic performance. In *Experimental Algorithmics*, LNCS 2547, pages 93–126, 2002.

[19] A. Mühlenfeld and F. Wotawa. Fault detection in multithreaded c++ server applications. *Electron. Notes Theor. Comput. Sci.*, 174(9):5–22, 2007. ISSN 1571-0661.

[20] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran. Spec omp2012 – an application benchmark suite for parallel systems using openmp. In *Proc. 8th Int. Conf. on OpenMP in a Heterogeneous World (IWOMP'12)*, pages 223–236, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] MySQL Web site. `http://www.mysql.com/`.

[22] mysqlslap load emulation client. `http://dev.mysql.com/doc/refman/5.5/en/mysqlslap.html`.

[23] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[24] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proc. 5th European Conference on Computer Systems (EuroSys'10)*, pages 335–348, 2010.

[25] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[26] J. M. Spivey. Fast, accurate call graph profiling. *Softw., Pract. Exper.*, 34(3):249–264, 2004.

[27] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 2006.

[28] Valgrind tool suite. `http://www.valgrind.org/info/tools.html`.

[29] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL*, pages 351–362. ACM, 2007.

[30] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447, 2004.

[31] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, pages 67–76. ACM, 2012.

[32] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proc. 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'09)*, pages 47–58, 2009.

[33] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, pages 263–271, 2006.